Safety Verification of Ada Programs in MURPHY

Stephen S. Cha
Nancy G. Leveson
Timothy J. Shimeall

Department of Information and Computer Science
University of California, Irvine
Irvine, CA 92717

Technical Report 87-23

Safety Verification of Ada Programs in MURPHY[1]

Stephen S. Cha Nancy G. Leveson Timothy J. Shimeall

Department of Information and Computer Science
University of California, Irvine
Irvine, CA 92717

September 28, 1987

Abstract

MURPHY is a experimental methodology, which will include an integrated tool set, for building safety-critical, real-time software. Although it is language independent, many safety-critical software projects are currently planning to use Ada. This paper presents the semantic templates for the verification of the safety of Ada programs using Software Fault Tree Analysis. An example is shown of applying the technique to an Ada program, and the tools in the MURPHY tool set to aid in this type of analysis are described.

1 Introduction

A system or subsystem may be described as *safety-critical* if there are potential consequences of using the system that are so serious that it cannot be used at all unless the probability of a high-cost event (an accident) occurring is very low. For example, a system is usually considered safety-critical if a run-time error or failure can result in death, injury, loss of equipment or property, or environmental harm. When computers are used to control safety-critical processes, there is a need to verify that the software will not cause or contribute to an accident. Until relatively recently, although computers were used in such potentially unsafe systems as aircraft, air traffic control, nuclear power, defense, and aerospace systems, a natural reluctance to add unknown and complex factors

[1]This work was partially supported by a MICRO grant co-funded by the State of California and Hughes Aircraft Company

to these systems kept computers out of most safety-critical loops. However, the potential advantages of using computers are now outweighing apprehension (some might say good sense), and both computer scientists and system engineers are finding themselves faced with potential liability and with new government standards and requirements (e.g., MIL-STD-882B Notice 1) that require certification of software safety to a degree not yet possible with current software engineering methods.

In safety-critical systems, it is not unusual to have reliability requirements of 10^{-5} to 10^{-9} probability of failure over a given period of time. This translates into requirements such as one failure per hundred years. Unfortunately, current software engineering technology cannot guarantee that such reliability is achieved for software (or, for that matter, even measured). Available evidence indicates that current software reliability figures are, at best, orders of magnitude less than required [2].

What can be done? One option is not to build these systems or not to use computers to control them. This option should be seriously considered by those making such decisions. The current rush to use computers to control nearly every type of device may involve a seriously unrealistic discounting of the potential risk. In some cases, reliability models based on totally unrealistic and unproven assumptions (e.g., [13] which is based on an assumption that has been experimentally shown to be false [5]) are used to justify the use of computers. In others, the evaluation of risk appears to be based solely on optimism. One reasonable conclusion is that in safety-critical systems where the potential risk must be very low, computers should not be used as the sole source of control without highly reliable back-up systems and independent (non-software) protection against software control errors.

There are, however, systems where a realistic risk/benefit tradeoff might conclude that computer control is justified. For example, it is easier to justify the use of fly-by-wire systems in a military fighter aircraft than in a commercial aircraft. In these cases, a non-

absolute approach to reliability may be possible. It is often not necessary for software to be completely correct in order for it to be safe; there are many types of failures possible in any complex system, with consequences varying from minor annoyance up to injury and death. For example, if the spacecraft software temporarily fails to archive some data for later analysis, the consequences are undesirable but not as serious as a failure involving the destruction of the spacecraft itself or non-fulfillment of the primary mission.

It seems reasonable to devise techniques that focus on those failures with the most serious consequences. Even if all failures cannot be prevented, it may be possible to ensure that the failures that do occur are of minor consequence or that even if a potentially serious failure does occur, the system will "fail safe" (i.e., fail in a manner that does not have unacceptable results).

This approach is useful under the following circumstances: (1) not all failures are of equal consequences and (2) there are a relatively small number of failures that are potentially serious. Under these circumstances, it is possible to augment traditional software engineering techniques that attempt to eliminate all errors with techniques that concentrate on potentially high-cost errors. These new techniques often involve a "backward" approach that starts with determining what are the unacceptable or high-cost failures of the software and then ensures that these particular failures do not occur or that their probability of occurrence is minimized. Another way of looking at this is that a "forward" analysis attempts to ensure that all possible reachable states of the system are correct whereas the goal of backward analysis is to ensure that particular incorrect states are not reachable. The latter is practical only under the above assumptions that there are a relatively small number of failures that are unacceptable and that these can be stated. In practice, this is usually the case even in complex systems. For example, this type of approach has been applied to nuclear power plants, commercial aircraft, and missile defense systems.

The UCI Safety Project is developing an experimental methodology called MURPHY that will include an integrated tool set for building safety-critical, real-time software. There are currently three main areas of research: (1) software hazard analysis and requirements specification techniques; (2) verification, validation, and assessment of safety; and (3) software design and run-time environments. This paper describes a technique for safety verification of software written in Ada[2]. Previously, Leveson and Harvey [6] developed a technique called Software Fault Tree Analysis and applied it to Pascal, and Leveson and Stolzy [7] demonstrated its use on the Ada rendezvous. This paper extends the technique to full Ada, provides an example of its use, and describes the tools currently completed and under development.

2 Fault Tree Analysis

A hazard is a set of conditions (state) that has an unacceptable risk of leading to an accident, given certain environmental conditions. System safety analysis involves determining the hazards of a system and then either verifying that the hazardous state cannot be reached or that the risk is acceptable. There have been several system safety engineering techniques developed to accomplish this. One of these, Fault Tree Analysis (FTA), was developed in the early 1960's to analyze the safety of electro-mechanical systems [12]. Software Fault Tree Analysis [6] was derived from and extends FTA to systems containing computers as subcomponents. In FTA, a hazard is specified, and the system is then analyzed in the context of its environment and operation to find credible sequences of events that can lead to this hazard. The fault tree itself is a graphic model of various parallel and sequential combinations of faults (or system states) that will result in the occurrence of the predefined undesired event. The faults can be events

[2]Ada is a registered trademark of the U.S. Department of Defense (Ada Joint Program Office).

that are associated with component hardware failures, human errors, or any other pertinent events that can lead to the undesired state. A fault tree thus depicts the logical interrelationships of basic events that lead to the hazard.

The basic procedure in FTA is to assume that the hazard has occurred and then to work backward to determine its set of possible causes. The root of the fault tree is the hazard and the necessary preconditions are described at the next level of the tree with either an AND or an OR relationship. Each subnode is expanded in a similar fashion until all leaves describe events of calculable probability or are unable to be analyzed for some reason.

Once the fault tree has been built down to the software interface, the high level requirements for software safety have been delineated in terms of software behavior (usually involving outputs or lack of outputs) that could adversely affect the safety of the system. Unsafe software behavior may result from:

- failing to perform a required function, i.e., never executing the function or not producing an answer,
- performing a function not required, i.e., getting the wrong answer, issuing the wrong control instruction, or doing the right thing but under inappropriate conditions (for example, activating an actuator inadvertently, too early, too late, or failing to cease an operation at a prescribed time),
- failing to enforce required sequencing, e.g., failing to ensure that two things happen at the same time, at different times, or in a particular order,
- failing to recognize a hazardous condition requiring corrective action
- producing the wrong response to a hazardous condition.

After the hazardous software behavior has been identified in the system fault tree, Software Fault Tree Analysis (SFTA) can be applied at the design or code level to identify safety-critical items and components, to detect software logic errors, to determine the conditions under which fault-tolerance and fail-safe procedures should be initiated and to guide in the placement and content of run-time checks to detect hazardous software states, and to facilitate effective safety testing by pinpointing critical functions and test cases. If used in conjunction with a system simulator, the interfaces of the software fault tree can be examined to determine appropriate simulation states and events.

3 Software Fault Tree Analysis

SFTA works backward from the critical control faults determined by the system fault tree through the program code or the design to the software inputs. The approach is similar to the backward reasoning used in formal axiomatic verification, but with a much more limited goal. That is, SFTA attempts to verify that the program will never allow a particular unsafe state to be reached, although it proves nothing about incorrect but safe states. Most real-time embedded systems have two goals: (1) accomplishing a mission or function, while (2) not causing harm in the process. SFTA is aimed at only the second goal.

There are several reasons to separate the verification of these two goals. First, different approaches may apply. Furthermore, partial verification may be less costly and therefore more practical. Finally, government licensing agencies are often concerned only with safety with respect to granting a license to use the system. For example, the Nuclear Regulatory Commission is concerned with the safety of nuclear power plants but not with whether they put out a certain amount of power or earn money for the utilities running them. Most safety-critical systems now have government licensing or certifica-

tion requirements. Potential liability concerns also provide incentive to perform safety verification.

Because the goal is to prove that the software will not do something, it is convenient to use proof by contradiction. In SFTA, it is hypothesized that the software has produced an unsafe control action, and it is shown that this could not happen since the hypothesis leads to a contradiction. If a path is found through the software and out into the controlled system or its environment that does not contain a logical contradiction, then the hazard is reachable and this needs to be considered in the design of the system. For example in a SFTA of a scientific satellite control program [6], it was found that the satellite could be destroyed if the input sensors detected two sun pulses within 64 ms of each other. The appropriate action in this case is to use run-time assertions to detect such conditions and simply to reject incorrect or unsafe input. In other cases it might be most appropriate to redesign the program, to initiate software recovery routines, or to redesign non-computer parts of the system.

SFTA has been successfully applied in several real software projects. Its success appears to be related to the fact that it forces the analyst to look at the software in a slightly different way. That is, usually the programmer is concerned with what the software is required to do. SFTA forces the programmer or analyst to consider what the software is *not* supposed to do. It also starts from a separate specification (the system fault tree) and therefore can possibly find errors in the software requirements specification. The process of working backward through the software and out into the environment allows identification of the most critical assumptions about the environment. In the satellite example cited above, the designers and programmers had been entirely unaware that the software was based on an assumption about the minimum timing interval of the incoming sun pulses.

4 SFTA Applied to Ada Programs

SFTA starts with the hazardous output and works backward through the code. The analysis proceeds based on statement-specific templates for generating the tree. Because the technique makes use of the semantics of the language in which the algorithm is specified, the templates may be different for each language. The basic templates for Ada statements are shown in figures 1 through 16. The templates were designed by examining the statement semantics as defined in the Ada Language Reference Manual[1] and by analyzing the causes of frequently-made programming errors. In each template, it is assumed that the statement caused the critical event, and the tree is constructed by considering how this might have occurred.

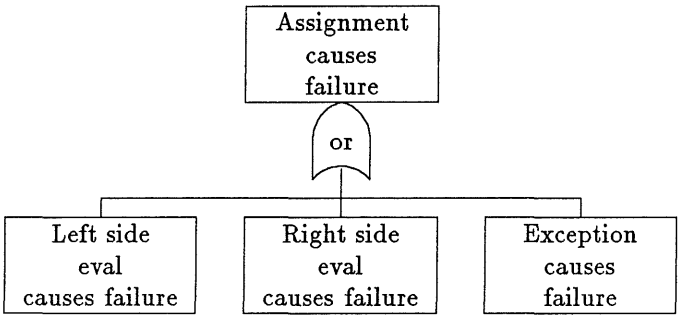

Figure 1: Assignment Template

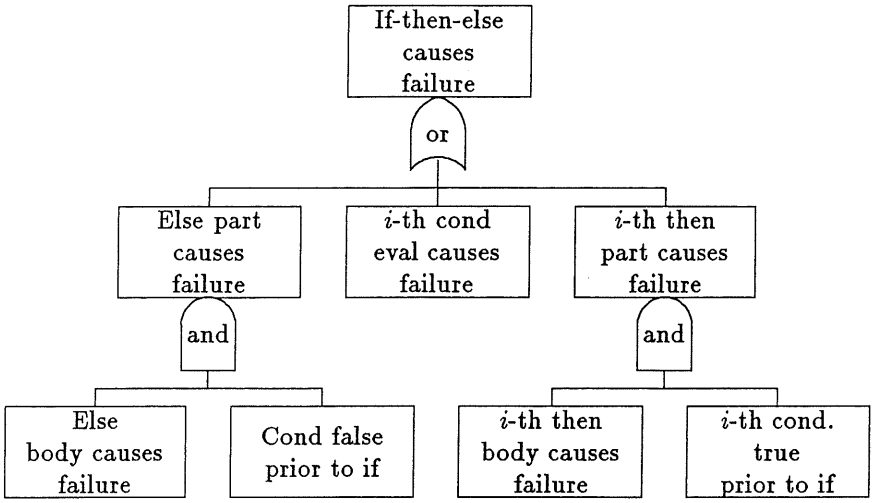

Figure 2: If–then–else Template

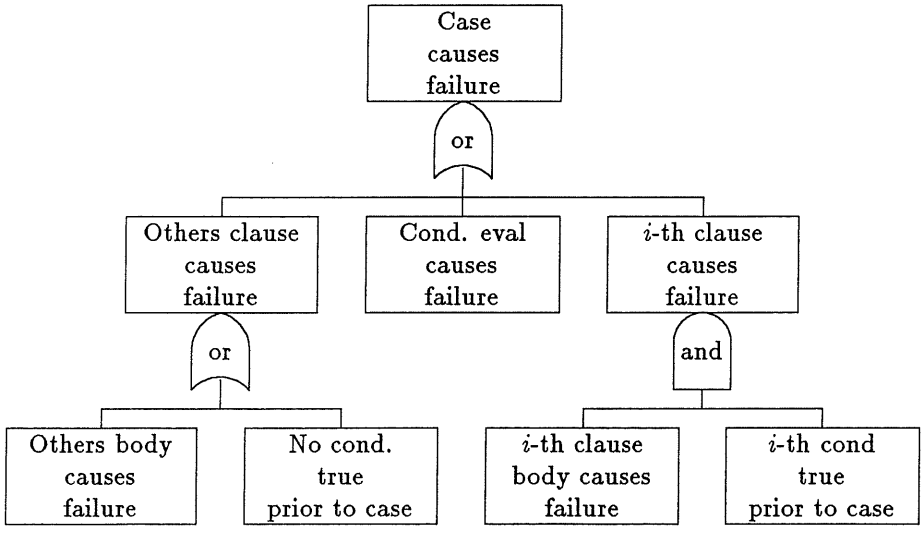

Figure 3: Case Template

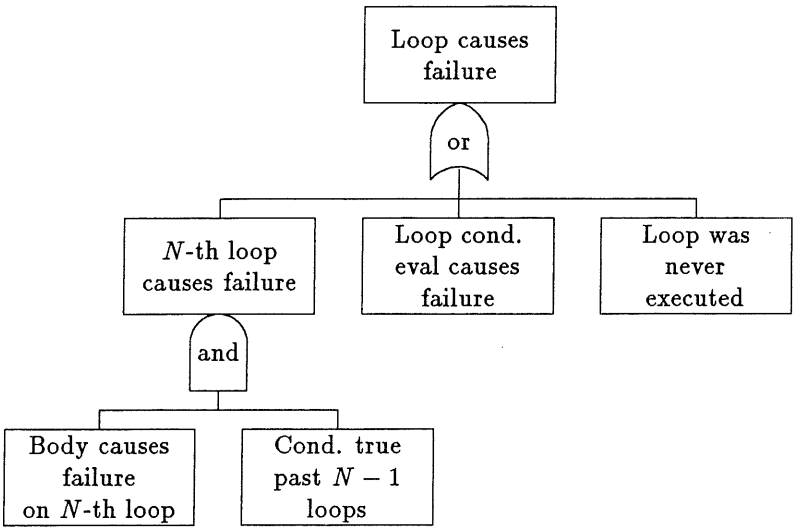

Figure 4: Loop Template

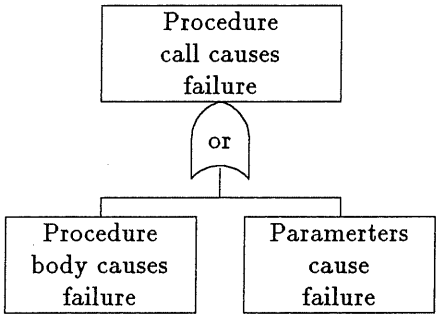

Figure 5: Procedure Call Template

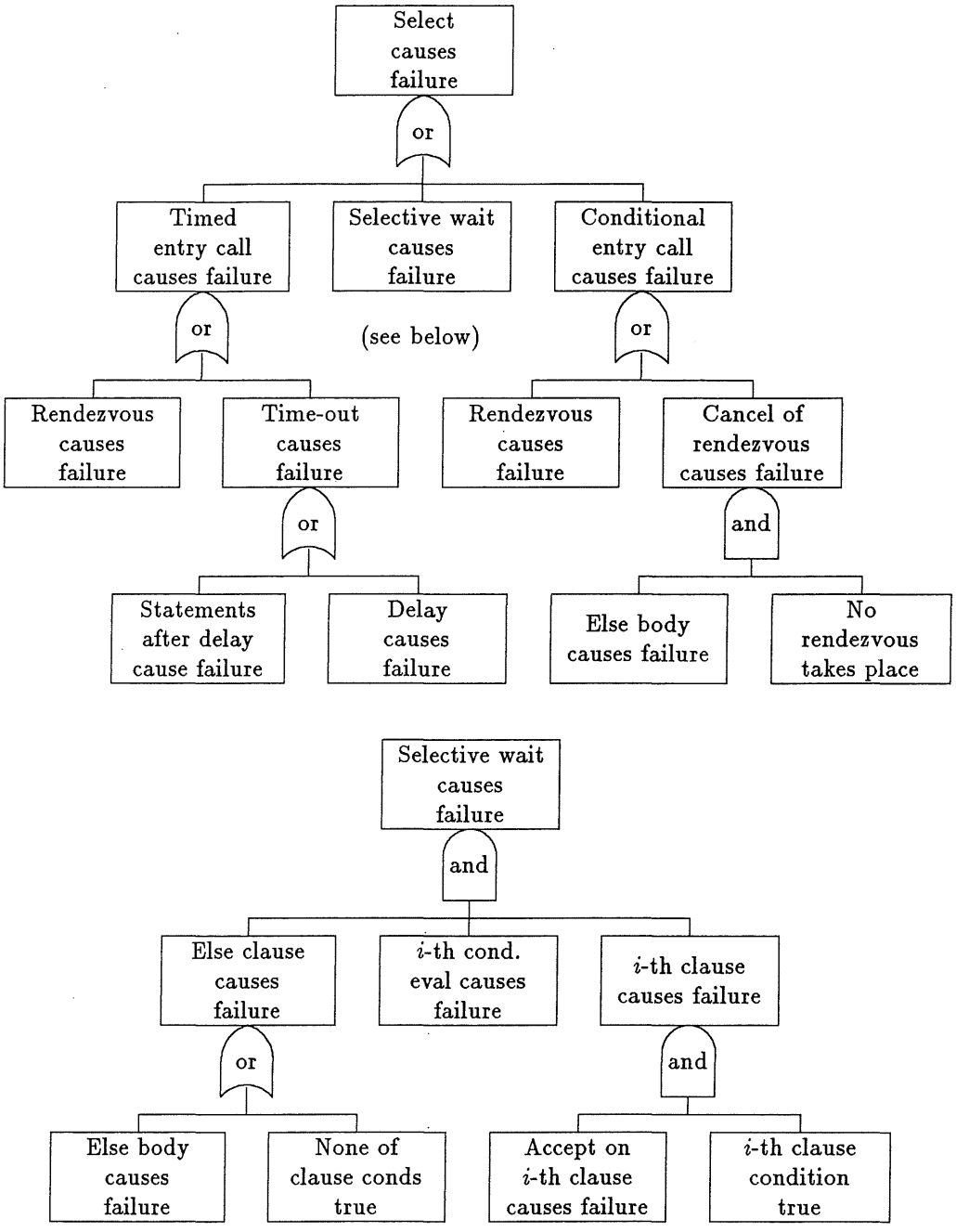

Figure 6: Select Template

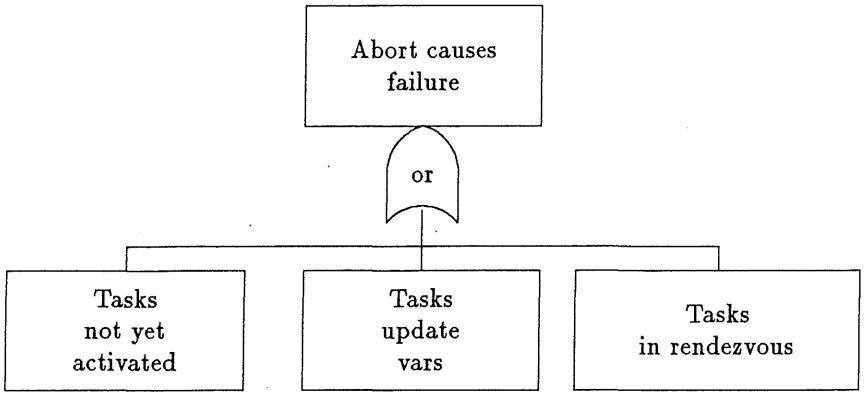

Figure 7: Abort Template

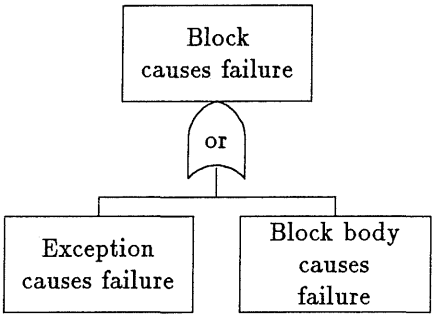

Figure 8: Block Template

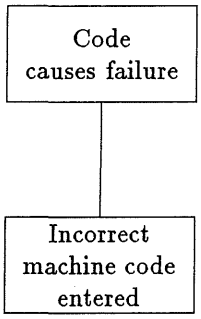

Figure 9: Code Template

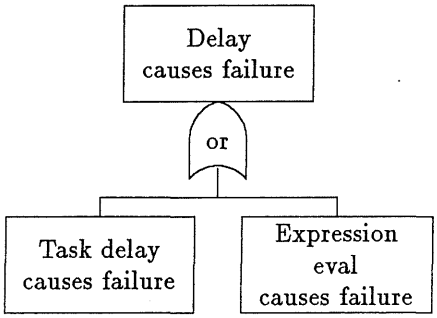

Figure 10: Delay Template

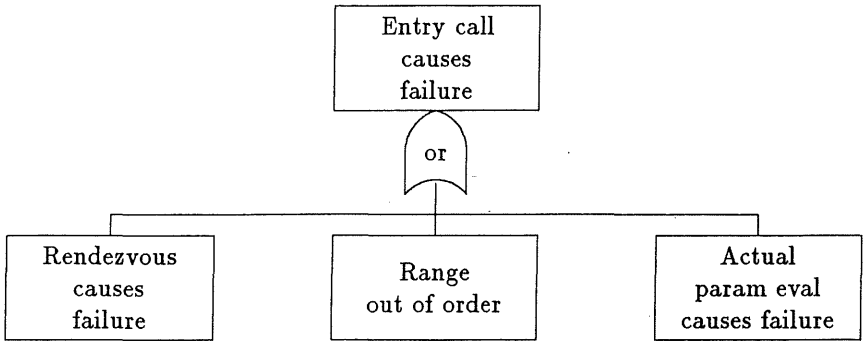

Figure 11: Entry Template

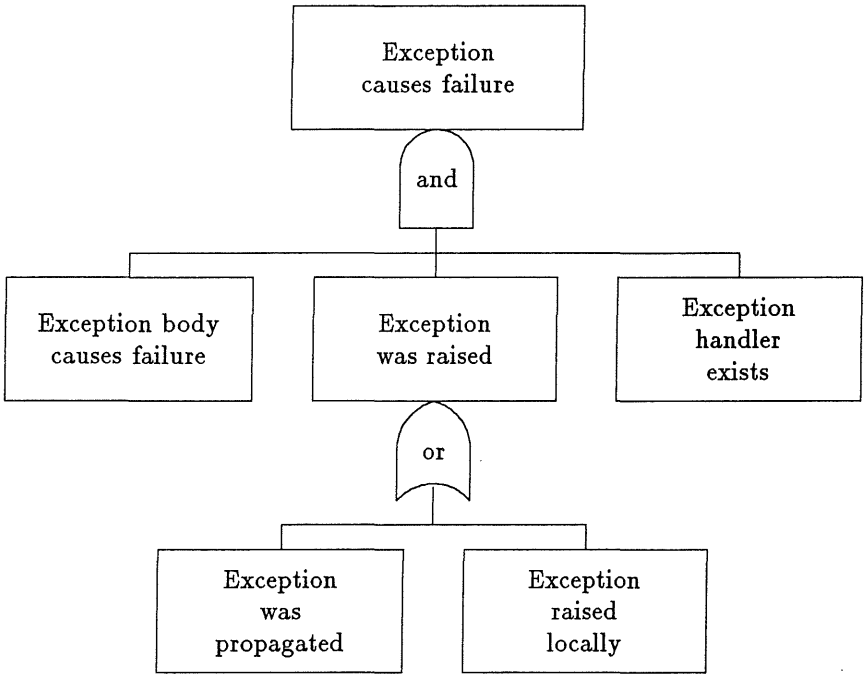

Figure 12: Exception Template

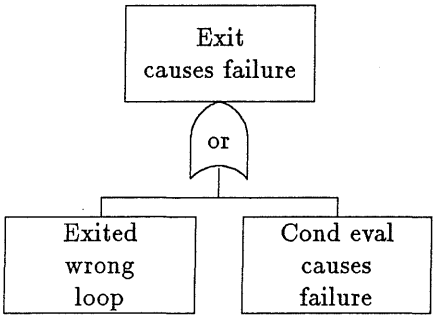

Figure 13: Exit Template

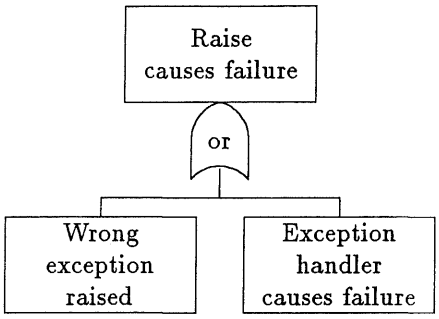

Figure 14: Raise Template

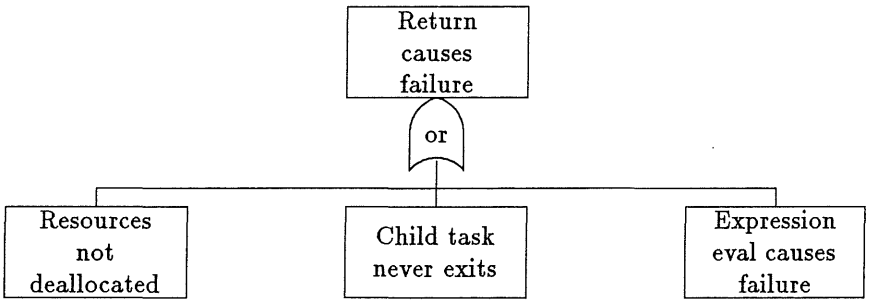

Figure 15: Return Template

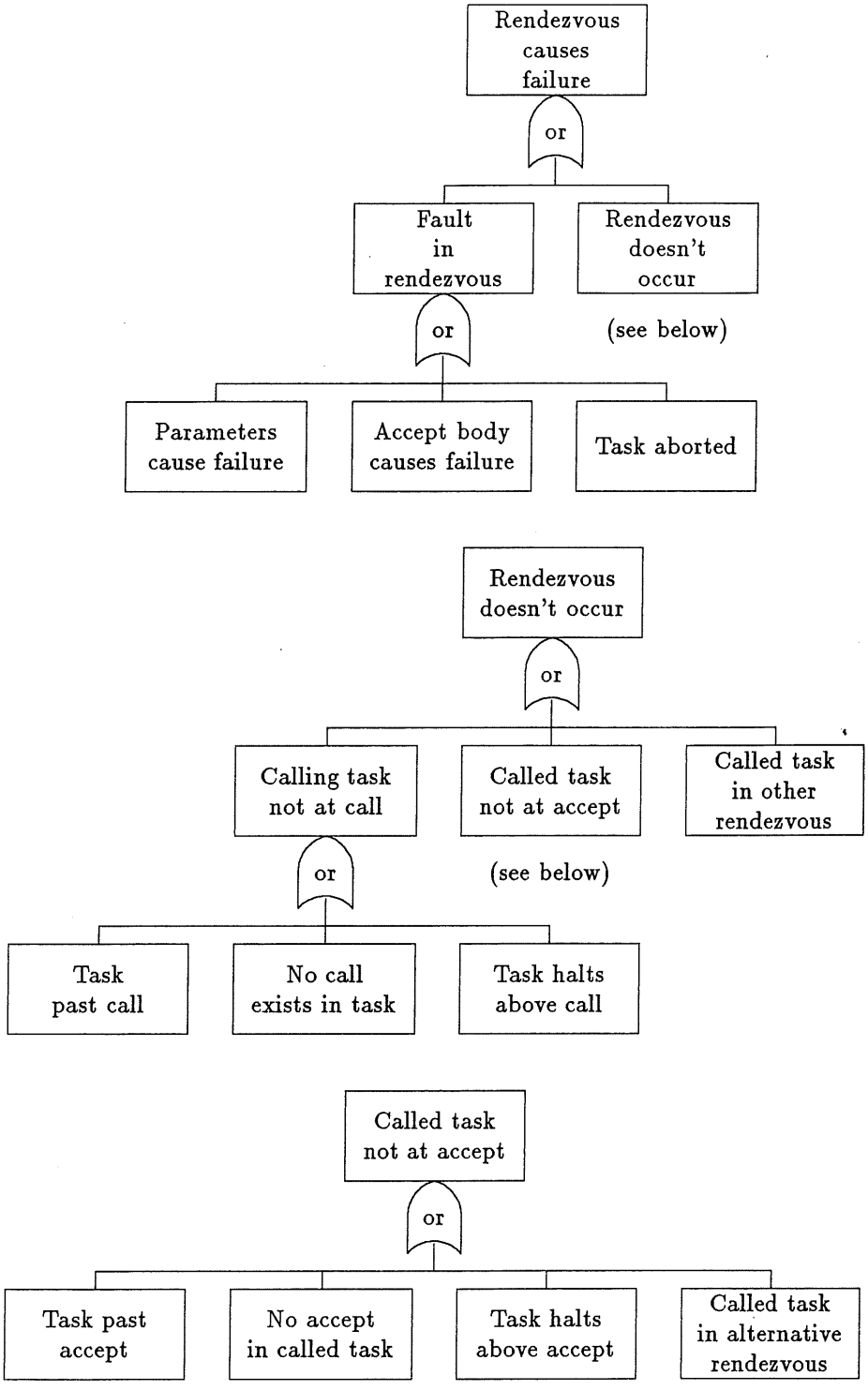

Figure 16: Rendezvous Template

As an example of how the templates are defined, consider the template for a rendezvous (figure 16). The event being analyzed (or 'failure') is caused by the rendezvous if either (1) the rendezvous not occurring could cause the failure or (2) the rendezvous does occur and a fault during the rendezvous could cause the failure. (If neither case holds, then the failure must have been caused by some prior statement, and the analyst must consider each prior statement in turn.) There are three ways a rendezvous could not occur: (1) the calling task may not be able to execute the entry call, (2) the called task may not be able to execute the accept, and (3) the called task may be able to execute the accept, but some task other than the calling task may have made an entry call on that accept and the called task proceeds with a rendezvous with this third task. If none of these conditions hold, then the rendezvous will occur, and the analyst must consider whether a fault in the rendezvous could cause the failure. To aid in this process, three cases are present in the template for the analyst to consider. In the first case, the values passed as parameters to the rendezvous are inappropriate and thus cause the failure. In the second case, the body of the accept statement contains a fault, which causes the failure. In the last case, the Ada Language Reference Manual notes that if a called task is aborted while it is in a rendezvous, then an exception will be raised in the calling task. The analyst needs to consider whether this exception could lead to the event being analyzed. If help is desired in that case, the analyst could consult the exception template (figure 12).

During the template development, we have made the following assumptions:

- The Ada program being analyzed is free from any syntax errors.
- The implementation of the underlying virtual machines (e.g., compiler) are perfect. Although this may not be true, it simplifies the templates. If desired, fault tree analysis can be applied to the underlying virtual machines (software or hardware)

to verify that they do not contribute to a hazard.

- The templates currently refer to faults made in the program body. The analysis of faulty declarations using fault trees is not included in this paper.

- Some statements, particularly **goto**, are difficult to analyze by a backward trace and are not included in this paper.

Perhaps the simplest way to explain the use of the templates is to illustrate it by analyzing a simple example problem:

> A traffic light control system at an intersection consists of four (identical) sensors and a central controller. The sensors in each direction detect cars approaching the intersection. If the traffic light currently is not green, the sensor notifies the controller so that the light will be changed. A car is expected to stop and wait for a green light. If the light is green already, the car may pass the intersection without stopping. The controller accepts change requests from the four sensors and arbitrates the traffic light changes. Once the controller changes the light in one direction (east-west or south-north) to green, it maintains the green signal for five seconds so that other cars in the same direction may pass the intersection without stopping. The light then changes from green to yellow and remains yellow for one second so that any car present in the intersection may clear. The light then turns to red while the light in the opposite direction turns green.

A sample Ada implementation of the problem is shown in figure 17. Due to the asymmetric nature of the Ada rendezvous (e.g., the called task does not know the identity of the calling task), an initialization (lines 17 through 19 and 45 through 47) is needed to assign a direction to each sensor. This direction is passed to the controller when

```
 1  procedure traffic is
 2    type direction is (east, west, south, north);
 3    type color is (red, yellow, green);
 4    type light_type is array (direction) of color;
 5    lights : light_type := (green, green, red, red);
 6    task type sensor_task is
 7     entry initialize (mydir : in direction);
 8     entry car_comes;
 9    end sensor_task;
10    sensor : array (direction) of sensor_task;
11    task controller is
12      entry notify (dir : in direction);
13    end controller;
14    task body sensor_task is
15      dir : direction;
16    begin
17      accept initialize (mydir : in direction) do
18        dir := mydir;
19      end initialize;
20      loop
21        accept car_comes;
22        if (lights(dir) /= green) then
23          controller.notify (dir);
24        end if;
25      end loop;
26    end sensor_task;
27    task body controller is
28    begin
29      loop
30        accept notify (dir : in direction) do
31          case dir is
32            when east | west =>
33              lights := (green, green, red, red);    delay 5.0;
34              lights := (yellow, yellow, red, red); delay 1.0;
35              lights := (red, red, green, green);
36            when south | north =>
37              lights := (red, red, green, green);    delay 5.0;
38              lights := (red, red, yellow, yellow); delay 1.0;
39              lights := (green, green, red, red);
40          end case;
41        end notify;
42      end loop;
43    end controller;
44  begin
45    for dir in east..north loop
46      sensor(dir).initialize (dir);
47    end loop;
48  end traffic;
```

Figure 17: Ada Implementation of Traffic Light

each sensor requests the controller to change the lights. When a car approaching the intersection is detected, the sensor for the corresponding direction executes line 21. The actual passing of the car through the intersection is assumed to begin when the program execution passes line 24 of the sensor task.

The trees in this paper make use of three of the standard fault tree symbols. A rectangle indicates an event that needs to be analyzed further. A diamond indicates an event which is not further analyzed, either because it is inapplicable to the statement being analyzed or because a contradiction is found. Finally, an oval indicates a condition normal to the operation of the system that contributes to the failure.

The application of SFTA to this program is illustrated by analyzing the event where two cars travelling from the north and east of the intersection are present in the intersection simultaneously. The analysis proceeds by finding the causes of the event and their relationships.

There could be many ways two cars travelling north and east could be in the intersection simultaneously (figure 18). The authors have chosen to examine the case where the north car enters before the east car clears the intersection[3]. After selecting the event to be analyzed, the next step in the analysis is to find recursively the possible subevents leading to the failure event. In this example, the authors have decided to explore (again, among several possibilities) the case where the `sensor(east)` task is in rendezvous with the controller task and the `sensor(north)` task bypassed the rendezvous point (null else). This implies that the signal was green when the car from the north approached the intersection, and it therefore entered the intersection without stopping. The tree indicates that the top event could happen if and only if both subevents occur.

[3]In fact, the authors are aware of several other failure modes for this program. We have chosen to explore only one possible case as an illustration here. Interested readers are encouraged to try to find the other failure modes.

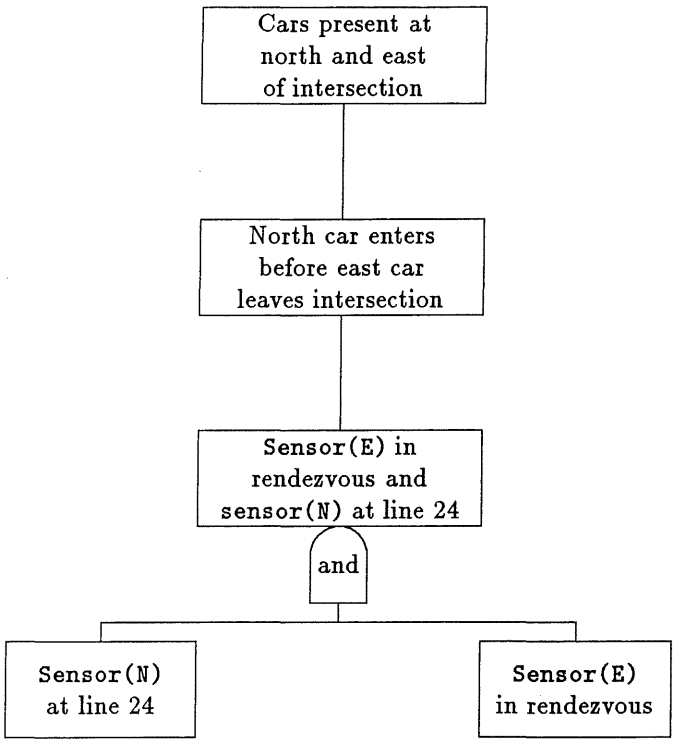

Figure 18: Top Level Fault Tree

In order to determine how the `sensor(north)` task bypasses the rendezvous point, it is necessary to trace the program backward from line 24 (figure 19). Since the immediately preceding statement is the **if** statement (lines 22 through 24), we attach the **if** template to the fault tree. A statement template is used here to offer the analyst suggestions as to how the specific statement might cause the fault. Since it is known that the `sensor(north)` task bypasses the rendezvous with the controller (then-part) and that the else-part does not have any statements, it is possible to immediately terminate the analysis along those two branches. The diamond symbol ("undeveloped event") is used to indicate this. The refinement of the leftmost branch is quite straightforward. For the task to bypass the rendezvous, the `lights(north)` must be green and this must happen before the east car clears the intersection for the top level event to happen.

The subevent "`sensor(east)` task in rendezvous" is analyzed in a similar way (fig-

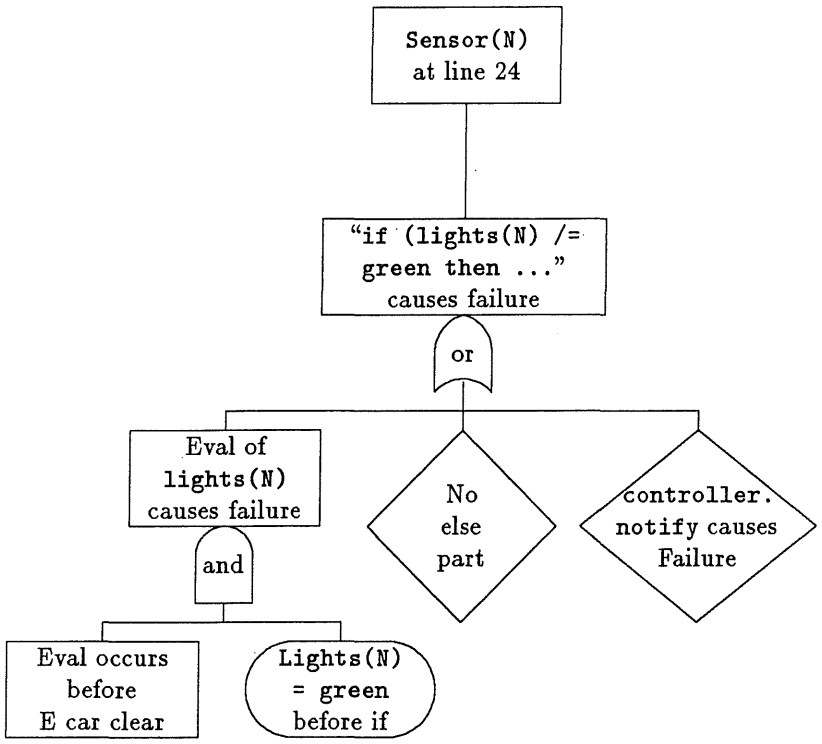

Figure 19: `Sensor(North)` at Line 24

ure 20). The template for the rendezvous is used here. Among the three possible causes, two are discarded immediately since the examination of the code indicates that there is no task abortion and that parameter evaluation (line 23) was not a direct cause of the event. While the former decision is obvious, the latter one represents a decision made by the analyst. The only branch left to explore further is the rightmost branch where the rendezvous body (line 33 through 35) causes the top event. Since the last statement in the body is not a direct cause of the event we are analyzing, the immediately preceding **delay** statement is analyzed. This represents a delay of 1.0 second in the yellow state as the light changes from green to red. The **delay** templates are used next. Since the possibility of the **delay** expression evaluation causing the failure can be excluded, the node "task delay caused failure" is examined next. The task delay is shown to be a cause of the event if the delay of 1.0 second is not enough time for a car to pass the intersection.

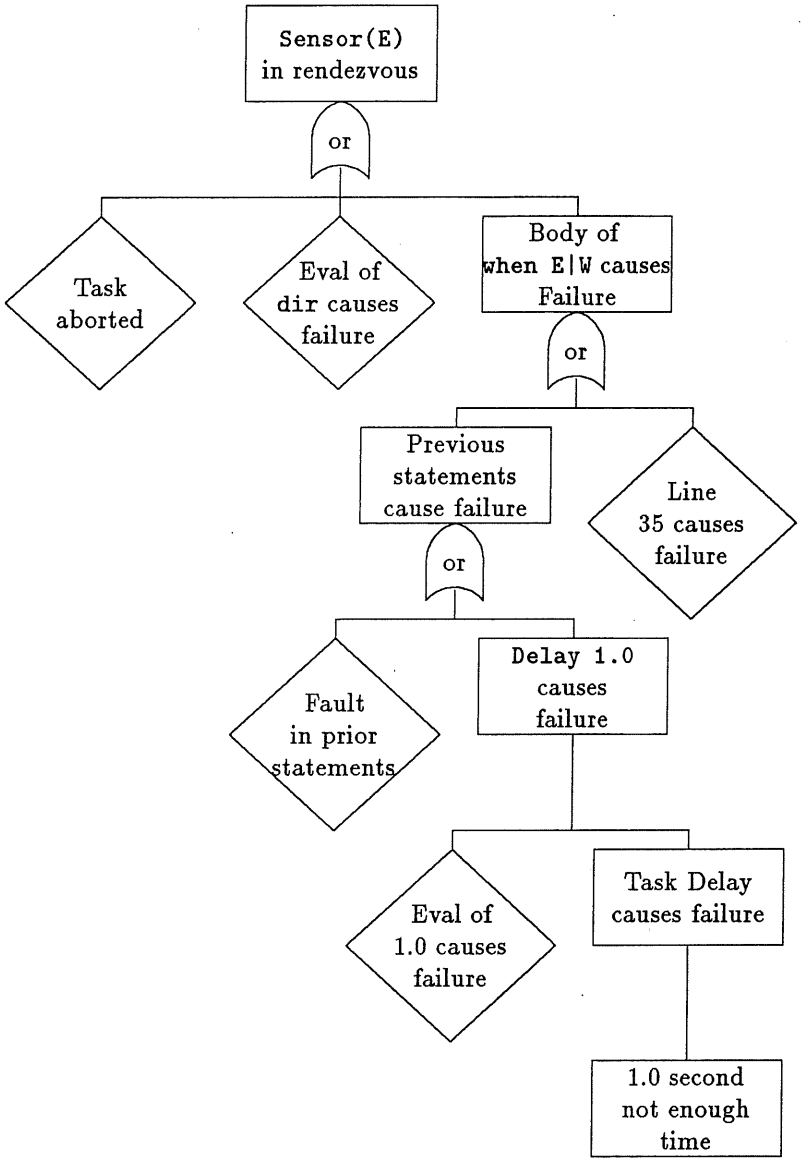

Figure 20: `Sensor(East)` in Rendezvous

In summary, the above fault tree analysis demonstrates that two cars could enter from north and east of the intersection simultaneously if the east car (which entered the intersection while the `sensor(east)` is in rendezvous with the controller) is unable to clear the intersection as the light changes from green to yellow to red. When the east light becomes red, the north light will become green simultaneously, allowing the north car to enter the intersection without stopping. The complete fault tree is shown in figure 21.

In general, a software fault tree has one or both of the following patterns:

1. A contradiction is found. The construction of the fault tree (at least for this path) can stop at this point since the logic of the software cannot cause the event. The example given above does not deal with the problem of failures in the underlying implementation of the software, but this is possible. There is, of course, a practical limit as to how much analysis can and need be done depending on individual factors associated with each project. It is possible to include assertions or exception conditions in the code to catch critical implementation errors at run-time if run-time software-initiated or software-controlled fault-tolerance and fail-safe procedures are feasible. Note that the software fault tree provides the information necessary to determine which assertions and run-time checks are the most critical and where they should be placed. Since checks at run-time are expensive in terms of time and other resources, this information is extremely useful.

2. The fault tree runs through the code and out to the controlled systems or its environment. The example fault tree above shows one possible path to the hazard, and changes are necessary to eliminate it.

The technique illustrated above can be used to analyze any Ada program subunit. Analysis of tasks and the rendezvous are performed as demonstrated in the example,

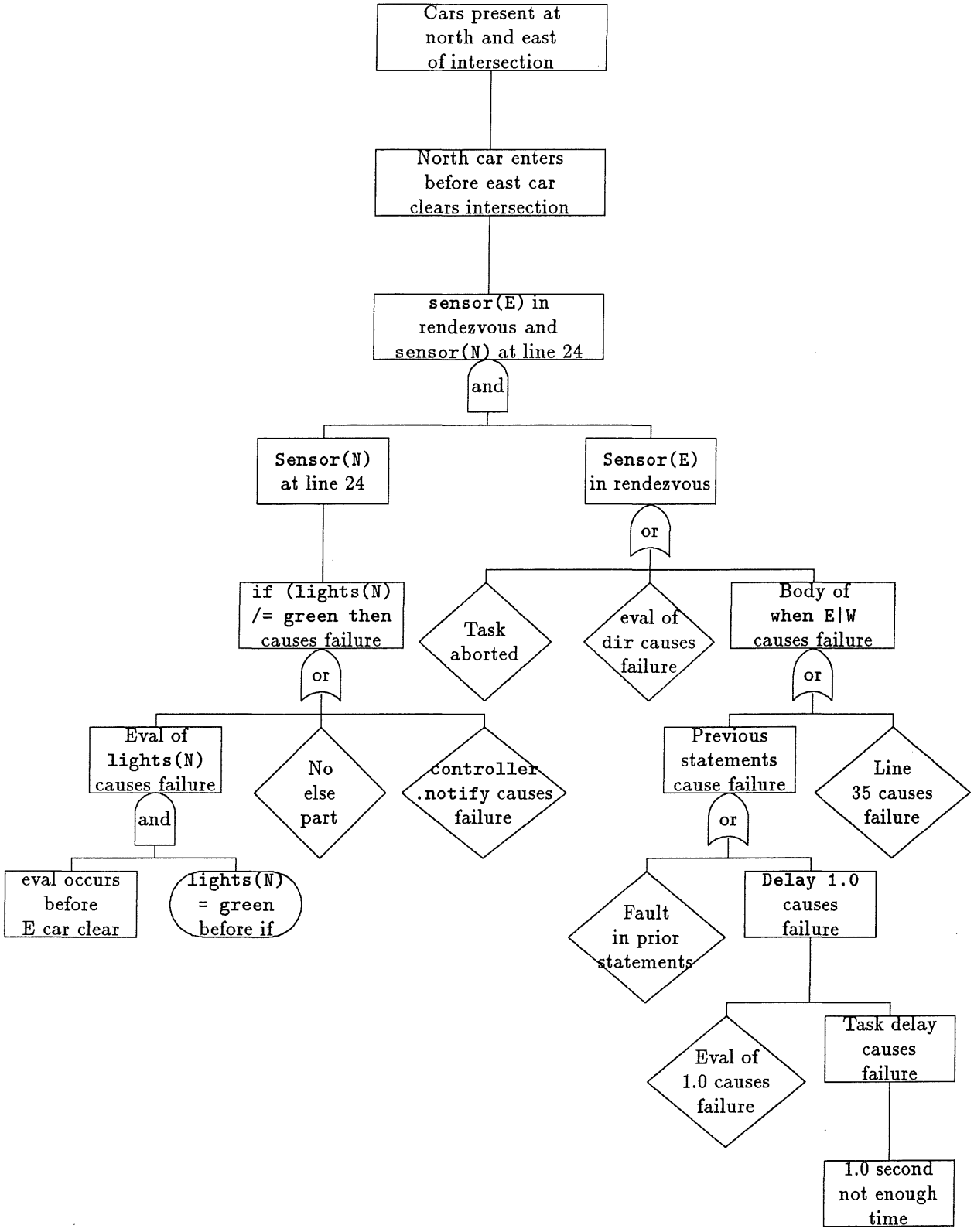

Figure 21: Final Fault Tree

assuming that all communication is through a rendezvous. Although communication using shared global variables is not prohibited in the language definition, its use in tasks is unsafe since simultaneous reading and writing could result in an undefined and most likely undesirable state. Similarly, use of the **goto** statement is discouraged since it changes the control flow arbitrarily.

Analysis of procedures and function bodies is straightforward (see [6] for a thorough explanation). Calls to procedures are analyzed using the procedure call template (figure 5). Since a procedure body is a sequence of simple or compound statements, it can be analyzed using the existing templates. Analysis of function calls is similar, and is part of the analysis of failures caused by expression evaluation. Packages and generics in Ada consist of subunit specifications and corresponding bodies. The bodies can be analyzed by examining the statements in each of the constituent packages, procedures and functions, as appropriate. It is necessary to include the initialization body of a package, and the effect of instantiation on a generic in the analysis.

The discussion above focuses on the application of the analysis procedure to Ada code. It is also possible to apply the same type of analysis to a design language. At the highest level of design, the analysis pinpoints the safety-critical components of the design. These can then be isolated for protection and further analysis, and fault-tolerance or other design features can be used. Careful design analysis has the potential for minimizing expensive verification procedures later in the development process.

Software Fault Tree Analysis also has important implications in the reuse of Ada components and packages. Accidents often arise from problems in the interfaces between components of a system. A recent software problem causing the death of three people [3] involved the reuse of software components. The interface between components of a system is composed of the assumptions the components make about each other. In terms of one component, its interface is the set of assumptions it makes about its environment.

A path through the component in a software fault tree can show the conditions in the environment under which that component will exhibit a certain behavior. If there is no such path through the software, then that behavior cannot occur. Even though reusable Ada packages can themselves be highly reliable, this alone does not preclude the possibility of problems arising in the interfaces of the reused packages.

5 Fault Tree Tools

5.1 Fault Tree Editor

The first tool in the MURPHY fault tree analysis tool set is an interactive screen-oriented fault tree editor[11,10]. This tool provides the analyst with the capability of creating or modifying fault trees in a structured manner. The editor performs no checking of the semantics of the fault tree generated. A version of the editor now in development will incorporate insertion of statement templates into the tree.

In figure 22 on page 24 a screen image from a session with the editor is shown. The analyst has just changed the connector (or 'gate') between the node labelled 'E in rendez., N at null' and its children by selecting the 'and' item from a pull-down menu. Other options available to the analyst allow modification of node shape, changes in its screen position or relationship to other nodes in the tree, adding or deleting nodes in the tree, and saving the tree for future reference or modification. The editor also provides graphic output of the fault tree by invoking the fault tree artist.

5.2 Fault Tree Artist

The fault tree artist takes the output format used by the fault tree editor and the fault tree generator and produces a graph of the tree in a standard format. Using a set of

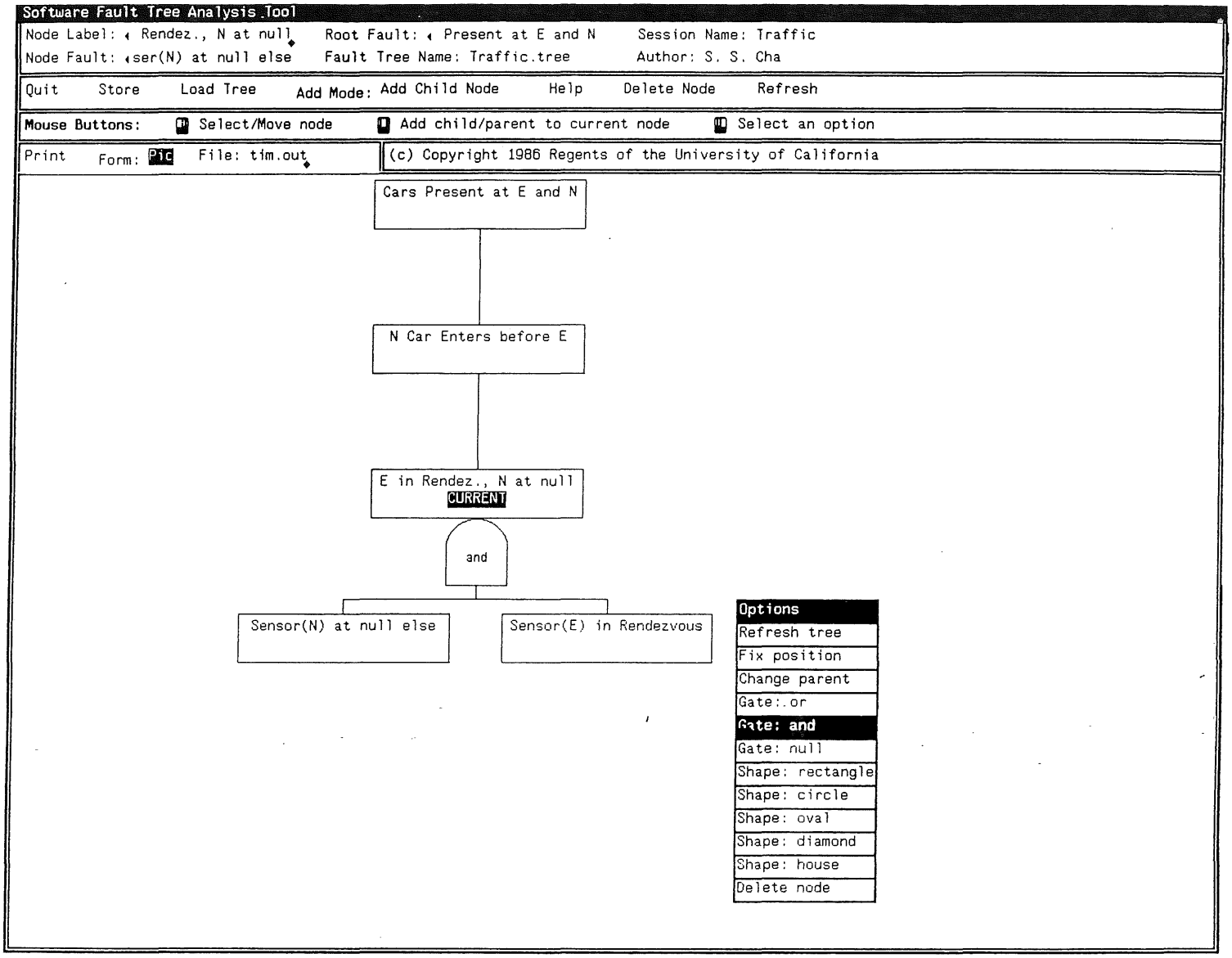

Figure 22: Sample Editor Session

layout algorithms[9], the artist handles the positioning of the tree on paper, dealing with off-page connections if the tree is too wide or too tall to fit on a single sheet. Three types of output are provided: WID, for wide line-printer forms (14 by 8.5 inches), LPT for narrow line-printer forms (8.5 by 11 inches) and PIC for output in the Pic[4] graphics language, which is translatable into either troff or TeX commands for final output. All fault tree illustrations appearing in this paper are produced by the fault tree artist.

5.3 Prototype Fault Tree Generator

A Fault Tree Generator is currently in development.[4] This tool takes an Ada program as input and, based on that program and interaction with the analyst, produces a fault tree or group of fault trees as output. The initial prototype of this tool when completed will have two parts, a translator from Ada into an intermediate form and a tree generator to turn the intermediate form into a fault tree. Once the tree is generated, it could be displayed using the fault tree artist or edited using the fault tree editor.

5.3.1 Translator

The first part of the fault tree generator is a translator that reads in an Ada program and translates it into a control-flow graph, annotated with the fault tree templates appropriate to each of the input statements. The translator is being built using the Lex/Yacc tools. As the graph is constructed, all of the statement-specific terms in the associated templates are replaced by the appropriate source fragment. For example, an Ada assignment statement would be translated into a control-flow graph node annotated with a copy of the template for assignment statements (see figure 1). In that copy of the template the words "left side" and "right side" would be replaced by the source text for

[4]Note to reviewer: This tool will be completed, if all goes as planned, by the time that the final version of this paper is due. The text can then be changed to the present tense.

the left and right hand sides of that assignment statement, respectively. This part of the translation task is being designed so that revisions of the templates can be easily incorporated (i.e., the templates are independent of the tool itself).

5.3.2 Prototype Tree Generation

Once the translator has produced the annotated flow graph, the tree generator traverses the flow graph and generates a fault tree. The initial version of this part of the fault tree generator is highly interactive. The analyst supplies the initial (root) fault and a program location, specifies which statements are to be expanded as branches of the fault tree, and enters additional information (such as loop invariants) as necessary. At any point in this process, the existing tree can be saved for modification, display, printing or later development. Future versions of this tool are planned that will incorporate what has been learned from initial use of the tool and will increase the automation available to the analyst (such as including weakest-precondition predicate transformation).

6 Conclusion

A procedure for safety verification of Ada programs has been presented along with an example and a description of some prototype tools to aid the analyst. Since fault tree analysis was originally used to analyze safety in electromechanical systems, the technique being developed has the advantage of being able to link together the software system and the controlled system at the interfaces of the two, allowing the system to be analyzed as a whole. It can be used at various levels and stages of software development and, although not shown in this paper, can include failures in the underlying computer hardware. The basic technique of software fault tree analysis has been applied successfully in real projects on assembly language programs [8] and other simple sequential programs. This

paper extends the analysis technique to include Ada language constructs. Since many of the future safety-critical programs are planned to be written in Ada, the extension is necessary if this type of safety verification is to be accomplished.

The reader is cautioned, however, in expecting too much from the technique. The analysis is very human-oriented, and therefore its success will depend on the ability of the analyst. The tools can aid the analyst, but do not guarantee success. SFTA, as defined here, is basically a form of structured walk-through. Its success in previous usage appears to be related to the fact that the analyst is forced to view the program in a different fashion than is common during development, and this increases the chance for finding errors. An analogy might be that when one vacuums a rug in one direction only, one is likely to pick up less dirt than if the vacuuming occurs in two opposite directions. That is, the more different ways a program is examined, the more likely that errors will be found. Programmers during development tend to concentrate on what they want the program to do; SFTA requires consideration of what the program should *not* do. However, the technique itself requires knowledge and experience on the part of the analyst and is not a substitute for any other type of verification and validation procedures. SFTA is just one part of the MURPHY methodology. In order to build software systems with acceptable risk, it will be necessary to make changes to and apply special procedures throughout the entire software development process.

7 Acknowledgements

The authors would like to acknowledge the help of Dr. John Goodenough of the Software Engineering Institute who provided information from a survey of errors found in Ada programs.

References

[1] ANSI/MIL-STD-1815A-1983. *The Programming Language Ada Reference Manual.* American National Standards Institute, February 1983.

[2] Janet R. Dunham and John C. Knight. *Production of Reliable Flight-Crucial Software.* Technical Report 2222, NASA, November 1981.

[3] Ed Joyce. Software bugs: a matter of life and liability. *Datamation*, 88–92, 15 May 1987.

[4] B. W. Kernighan. Pic – a language for typesetting graphics. *Software – Practice and Experience*, 12(1):1–21, 1982.

[5] John C. Knight, Nancy G. Leveson, and Lois D. St.Jean. A large scale experiment in n-version programming. In *International Symposium on Fault Tolerant Computing Systems*, pages 135–139, June 1985.

[6] Nancy G. Leveson and Peter R. Harvey. Analyzing software safety. *IEEE Transaction on Software Engineering*, SE-9(5):569–579, September 1983.

[7] Nancy G. Leveson and Janice L. Stolzy. Safety analysis of ada programs using fault trees. *IEEE Transactions on Reliability*, R-32(5):479–484, December 1983.

[8] James W. McIntee Jr. *Fault Tree Techniques As Applied to Software (Soft Tree).* Technical Report, USAF, March 1983.

[9] E. M. Reingold and J. S. Tilford. Tidier drawing of trees. *IEEE Transaction on Software Engineering*, SE–7(2):223–228, March 1981.

[10] C. Rolandelli, T. J. Shimeall, C. Genung, and N. Leveson. *Software Fault Tree Analysis Tool User's Manual.* Technical Report 86–06, University of California, Irvine, February 1986.

[11] J. L. Stolzy. Software fault tree analysis tool. University of California, Irvine, December 1984.

[12] W.E. Vesely, F.F. Goldberg, N.H. Roberts, and D.F. Haasl. *Fault Tree Handbook.* U.S. Nuclear Regulatory Commission, nureg–0492 edition, January 1981.

[13] L. J. Yount, K. A. Lievel, and B. H. Hill. Fault effect protection and partitioning for fly-by-wire/fly-by-light avionics systems. In *AIAA Computers in Aerospace V Conference*, pages 275–284, AIAA, Long Beach, CA, October 1985.